Centro Universitario de Ciencias Exactas e Ingenierías.

Ciencias de la computación

Seminario de Uso, Adaptación y Explotación de Sistemas Operativos

Becerra Velázquez Violeta del Rocío

Hernandez Lomelí Diego Armando

219750396

INNI- Ingeniería en informática

D02

Actividad de aprendizaje 10

2.3 Exclusión mutua

2023/10/11

Tabla de contenido

Tabla de imágenes

Introducción:

La exclusión mutua es un problema derivado de la concurrencia, en donde se refiere a las situaciones en las que dos o más procesos puedan coincidir en el acceso a un recurso compartido o, dicho de otra forma, que requieran coordinarse en su ejecución. Para evitar dicha coincidencia, el sistema operativo ofrece mecanismos de arbitraje que permiten coordinar la ejecución de los procesos. Para ello cuenta con diferentes soluciones tanto por hardware, por el sistema y por software.

Listado de soluciones

Liste cada una de las diferentes soluciones para hacer cumplir la exclusión mutua.

- **Monitores:**

Los semáforos son un modelo en programación que permite la ejecución de procesos a través de 1 único camino controlado por el monitor contenedor, esta estrategia debe ser implementada por el programador en todo sentido, lo hace altamente personalizable pero a la vez, susceptible a fallos si no se tiene el cuidado suficiente para realizar la exclusión mutua, es relativamente fácil de aplicar conteniendo pocos elementos principales por mantener dentro del monitor(datos) y funcionalidades limitadas en cantidad para gestionar internamente los procesos (su reanudación o bloqueo).

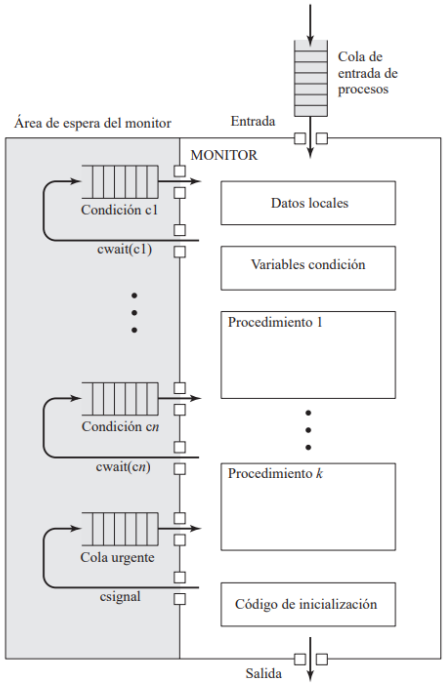


Figura 5.15. Estructura de un monitor.

Ilustración 1 figura 5.15 capitulo 5 "Concurrencia, exclusión mutua y sincronización"

En la figura adyacente se muestra la estructura del monitor. Contiene un único punto de entrada protegido que, en caso de recibir más de un proceso, los que lleguen después del primero serán colocados en una cola de espera que serán colocados como bloqueados.

Los procesos ingresados en el monitor pueden bloquearse a sí mismos con la función *cwait* que lo añadirá a la cola de espera como bloqueado y retomado después con función *cwait.*

- **Semáforos:** Variable numérica entera que define 3 operaciones diferentes
  - Inicialización: se puede inicializar a un valor no negativo.
  - Método *semWait*: decrementa el valor del semáforo que al ser negativo bloquea al proceso ejecutado en semWait.
  - Método *semSignal* incrementa el valor del semáforo. SI el valor es menor o igual a 0 entones se desbloquea un proceso bloqueado por *semWait.*
- **Mutex:**

Tipo de semáforo que como su nombre lo indica, su valor oscila entre 1 y 0 para la ejecución de procesos. Así mismo cuenta con 3 operaciones propias.

- Inicialización: puede ser declarado en 0 o 1.
- Método *semWaiB:* comprueba el valor del semáforo. Si el valor esta asignado a 0 el proceso será bloqueado. Si el valor esta asignado a 1 permite al proceso continuar con su ejecución, pero cambiando el semáforo a 0.
- Método *semSignalB*: comprueba si existe un proceso bloqueado en el semáforo. En caso de existir, lo desbloquea. Si no hay procesos bloqueados, el semáforo será asignado a 1.

- **Dekker:**

Solución que se describe como un sistema de turnos para la ejecución de procesos que requieran acceder a su sección critica. La intención de este algoritmo es hacer que se compruebe constantemente el estado de procesos **P1** y **P0**, el chequeo constante del estado permitirá saber si el proceso requiere entrar a su sección critica siendo que, si el estado de **P1** es falso, **P0** intentará entrar a su sección critica (solo cuando el estado de **P0** sea verdadero) pero si **P1** quiere acceder a esta sección, entonces **P0** deberá consultar su turno que será 0. Cuando es el turno de uno de los procesos estos deben insistir constantemente en la obtención del recurso a la vez que se comprueba el estado del proceso en conjunto, cuando cualquiera de los procesos ya salió de su sección critica, se establece el estado a falso y coloca el valor del turno a 1 o 0 respectivamente para que el otro proceso pueda insistir en su lugar.

Esta solución es resultado de 2 de sus 4 variaciones anteriores que combina la primera solución y la última simultáneamente donde la primera garantizaba exclusión mutua pero su ritmo y velocidad esta determinada por el proceso más lento en la lista y el último que aun sin tener procesos sin interbloqueo existía la posibilidad de dejar un proceso en espera a un evento que puede no ocurrir.

- **Peterson:**

El algoritmo de Peterson llega como una alternativa para el algoritmo de *Dekker*, que, aun resolviendo la exclusión mutua, generaba un programa extenso y difícil de seguir y comprobar.

- **Paso de mensajes:**

El paso de mensajes hace que los procesos interactúen entre ellos, el paso de mensajes debe satisfacer 2 condiciones, ser sincronizados y realizar comunicación. Esto debido a que con la sincronización se logra la exclusión mutua en cooperación y así mismo, es posible que los procesos necesiten comunicarse enviándose información relevante para su resolución.

Identificación de ejemplos de las soluciones.

- **Monitores:**

Utilizando el problema *productor/consumidor* tomaremos al monitor como un módulo. En esta situación se agregarán 2 variables (iniciadas en la construcción *cond*) una de ellas es *nolleno* que sirve para saber que hay espacio para añadir por lo menos un carácter y *novacio* que indica que por lo menos hay 1 carácter.

La definición de este problema indica que el productor solo puede añadir caracteres usando el método *anyadir* encapsulado en el monitor. Se comprueba que exista espacio disponible. Si no, se detendrá el proceso se bloquea, ahora cualquier proceso ya sea productor o consumidor puede entrar en el monitor. Cuando el buffer ya no se considere lleno, se extraerá el proceso bloqueado anteriormente y se retomará su ejecución. La función consumidor tiene una descripción similar ante el problema.

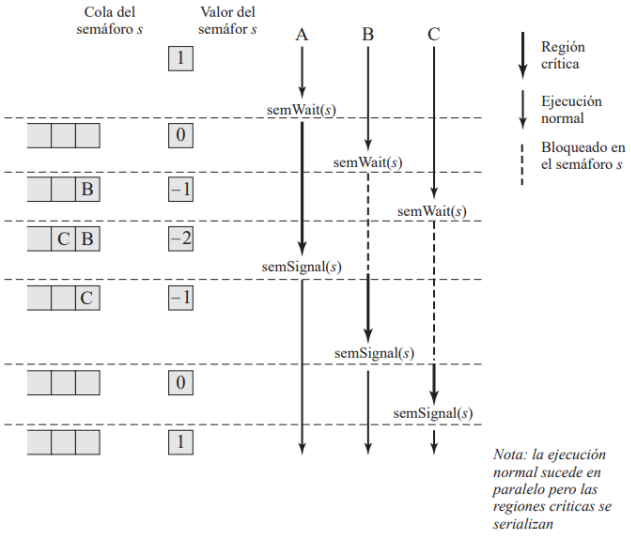
Con este ejemplo se hace notar la división de responsabilidades en los monitores comparado con los semáforos. Los monitores se imponen a sí mismos una exclusión mutua, no permite que ni el productor ni el consumidor accedan al buffer a la vez. Pero a su vez, el programador debe hacerse caso de utilizar *cwait* y *csignal* en el código del monitor. En este caso la exclusión mutua y la sincronización caen en manos del programador.

```
/* programa productor consumidor */
monitor buferacotado;
char buffer[N];                                   /* espacio para N datos */
int dentro, fuera;                                /* punteros al buffer */
int cuenta;                                       /* número de datos en el buffer */
cond nolleno, novacio;                            /* variables condición para sincronizar */
void anyadir (char x)
{
    if (cuenta == N)
        cwait(nolleno);                           /* buffer lleno, evitar desbordar */
    buffer[dentro] = x;
    dentro = (dentro + 1) % N;
    cuenta++;                                     /* un dato más en el buffer */
    csignal(novacio);                             /* retoma algún consumidor en espera */
}
void extraer (char x)
{
    if (cuenta == 0)
        cwait(novacio);                           /* buffer vacío, evitar consumo */
    x = buffer[fuera];
    fuera = (fuera + 1) % N;
    cuenta—;                                      /* un dato menos en el buffer */
    csignal(nolleno);                             /* retoma algún productor en espera */
}
{                                                 /* cuerpo del monitor */
    dentro = 0; fuera = 0; cuenta = 0;            /* buffer inicialmente vacío */
}
```

Ilustración 2 ejemplo de implementación de monitores con el problema del productor consumidor

- **Semáforos:**

En la ejecución del semáforo digamos que contiene 1 cola con 3 procesos. El primer proceso en entrar es **A** que se ejecuta inmediatamente por estar inicializado el semáforo en 1, después de utilizar la función *semWait* se reduce el valor del semáforo a 0.



Después entra **B** decrementando de nuevo al semáforo asignándolo a -1, esto bloquea al proceso **B** que ahora está en espera de ser desbloqueado. Acto seguido entra **C** y se repite el procedimiento.

Después de varias iteraciones, el proceso **A** termina su ejecución ejecutando *semSignal* y aumentando el valor del semáforo a -1 de nuevo (por la previa entrada de **C** al semáforo), el mismo proceso desbloquea a cualquier otro que estuviera en espera y con ello da paso a **B** para ser ejecutado. Consecuentemente **B**

termina su ejecución y aumenta el valor del semáforo ahora a 0 y dando paso a C para ser ejecutado.

La exclusión mutua se da al mantener a otros procesos a la espera de ser atendidos mientras uno de ellos en su sección critica se está ejecutando.

A continuación, se muestra un ejemplo de implementación de los semáforos no binarios.

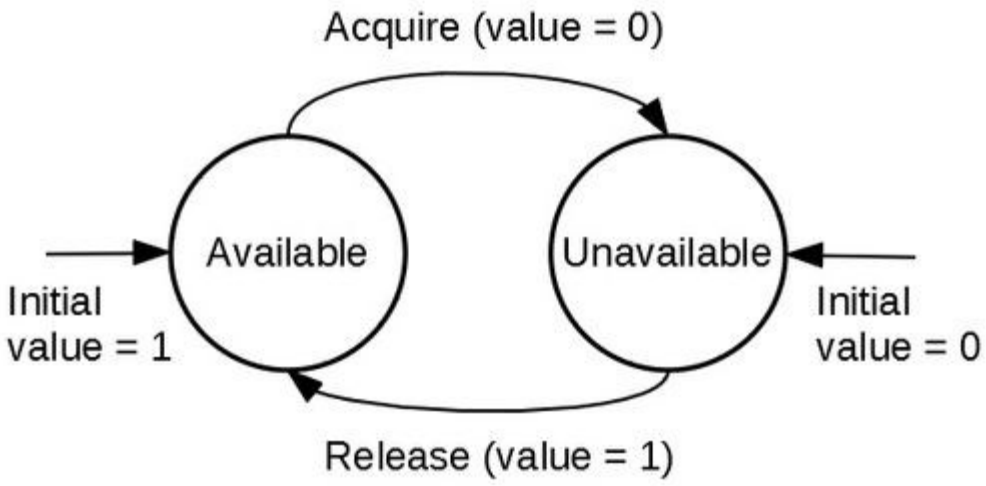
```
struct semaphore {
    int cuenta;
    queueType cola;
}
void semWait(semaphore s)
{
    s.cuenta—;
    if (s.cuenta < 0)
    {
        poner este proceso en s.cola;
        bloquear este proceso;
    }
}
void semSignal(semaphore s)
{
    s.cuenta++;
    if (s.cuenta <= 0)
    {
        extraer un proceso P de s.cola;
        poner el proceso P en la lista de listos;
    }
}
```

Figura 5.3. Una definición de las primitivas del semáforo.

Ilustración 3 ejemplo de implementación de semáforo no binario

- **Mutex:**

Un mutex al ser un semáforo tiene una metodología muy similar a los semáforos por estar basados en estrategias similares, en la imagen adyacente vemos que a diferencia de un semáforo convencional que, un mutex verifica solo entre 2 posibles valores que actúan como bandera y que cada chequeo correspondiente se encarga alternar el estado del semáforo, este bloqueo de primera mano ya interrumpe la ejecución de cualquier otro proceso entrante y lo libera únicamente cuando termina. En si mismo representa un ciclo de bloqueo y liberación más simplificado que un semáforo no binario.



Una implementación básica para un mutex es la siguiente.

```
struct binary_semaphore {
    enum {cero, uno} valor;
    queueType cola;
};
void semWaitB(binary_semaphore s)
{
    if (s.valor == 1)
        s.valor = 0;
    else
        {
            poner este proceso en s.cola;
            bloquear este proceso;
    }
}
void semSignalB(binary_semaphore s)
{
    if (esta_vacia(s.cola))
        s.valor = 1;
    else
    {
            extraer un proceso P de s.cola;
            poner el proceso P en la lista de listos;
    }
}
```

Figura 5.4. Una definición de las primitivas del semáforo binario.

Ilustración 4 ejemplo implementación de mutex

- **Dekker:** A continuación, se muestra un ejemplo de implementación del algoritmo Dekker.

```
boolean estado[2];
int turno;
void P0( )
{
    while (true)
    {
        estado[0] = true;
        while (estado [1]
            if (turno == 1)
        {
            estado[0] = false;
            while (turno == 1)
                /* no hacer nada */;
            estado[0] = true;
        }
        /* sección crítica */;
        turno = 1;
        estado[0] = false;
        /* resto */;
    }
}
void P1( )
{
    while (true)
    {
        estado[1] = true;
        while (estado[0])
        if (turno == 0)
        {
            estado[1] = false;
            while (turno == 0)
                /* no hacer nada */;
            estado[1] = true;
        }
        /* sección crítica */;
        turno = 0;
        estado[1] = false;
        /* remainder */;
    }
}
void main ( )
{
    estado[0] = false;
    estado[1] = false;
    turno = 1;
    paralelos (P0, P1);
}
```

Ilustración 5 ejemplo implementación de dekker

- **Peterson:** Ejemplo de implementación de algoritmo de Peterson.

```
boolean estado[2];
int turno;
void P0( )
{
    while (true)
    {
        estado[0] = true;
        turno = 1;
        while (estado [1] && turno == 1)
            /* no hacer nada */;
        /* sección crítica */;
        estado [0] = false;
        /* resto */;
    }
}
void P1( )
{
    while (true)
    {
        estado [1] = true;
        turno = 0;
        while (estado [0] && turno == 0)
            /* no hacer nada */;
        /* sección crítica */;
        estado [1] = false;
        /* resto */
    }
}
void main( )
{
    estado [0] = false;
    estado [1] = false;
    paralelos (P0, P1);
}
```

Figura A3. Algoritmo de Peterson para dos procesos.

Ilustración 6 ejemplo de implementación de agoritmo de Peterson para 2 procesos simultaneos

- **Paso de mensajes:**

A continuación, se muestra una implementación de paso de mensajes para la exclusión mutua.

En la implementación notemos que se genera una rutina para la función **P** que inicialmente espera un mensaje de un buzón involucrando en ámbito local de la función a un mensaje. Durante el recibimiento del mensaje se accede a sección critica mientras que después del envío se completa el resto del procedimiento, esto sucede para una cantidad **n** de procesos.

```
/* programa exclusión mutua */
const int n = /* número de procesos */;
void P(int i)
{
    message carta;
    while (true)
    {
        receive (buzon, carta);
        /* sección crítica */;
        send (buzon, carta);
        /* resto */;
    }
}
void main()
{
        create_mailbox (buzon);
        send (buzon, null);
        paralelos (P(1), P(2), . . ., P(n));
}
```

Figura 5.20. Exclusión mutua usando mensajes.

Ilustración 7 implementación de mensajes para exclusión mutua

Síntesis

Una vez concluida la investigación realice una síntesis donde incluya que es, sus diferentes soluciones y ejemplos de cada uno.

- **Monitores:**

Son una construcción de programación que son similares a los semaforos pero facilitan la gestión de acceso. Son implementaciones con bibliotecas que dan pie a escribir las señales o cerrojos desde cualquier objeto creado.

Las características principales son:

- Las variables locales son solo accesibles por los procedimientos del monitos, no son accesibles a otros contextos.
- Un proceso de entrada en el monitor invoca a uno de sus procedimientos.
- Solo se puede ejecutar un proceso a la vez dentro del monitor, otros procesos que invoquen al monitor se bloquean esperando disponibilidad del monitor.

Estas características facilitan la exclusión mutua al permitir ejecutar un único monitor a la vez, así mismo, los datos dentro del monitor se les puede considerar como protegidos, esta política de ejecución puede acarrear problemas de rendimiento si el proceso además de requerir el recurso necesita tiempo adicional o esperar un evento externo, para ello se deben incluir herramientas para sincronización.

Si un proceso dentro de esa condición existe en el monitor no solo se debe bloquear, también es importante liberar al monitor para que pueda ejecutar otro proceso en espera a que el primero satisfaga su condición que lo bloquea para volver a ser ejecutado cuando el monitor esté disponible. Esta sincronización se debe realizar a través de variables de condición que están encapsuladas en el monitor y que deben ser manipuladas en los métodos.

- cwait(c): que suspende la ejecución con a condición “c” y que libera el monitor.
- Csignal(c): Retoma la ejecución de un proceso bloqueado por “cwait” en la misma condición. Si existe previamente un proceso se retoma uno de ellos, si no hay ninguno, no se realiza ninguna operación.

Semáforos interactúan con sus variables como un cerrojo a los datos utilizando variables condición para permitir el acceso, son una representación del lenguaje lingüístico pero especificado de manera abstracta. Dicho de otro modo, mantienen el foco de protección en un único proceso como si un recepcionista en un establecimiento hotelero se tratase.

- **Semáforos:**

Variable numérica entera que dentro de una estructura que define 3 operaciones diferentes:

- Inicialización: se puede inicializar a un valor no negativo.
- Método *semWait*: decrementa el valor del semáforo que al ser negativo bloquea al proceso ejecutado en semWait.
- Método *semSignal* incrementa el valor del semáforo. SI el valor es menor o igual a 0 entones se desbloquea un proceso bloqueado por *semWait.*

Existe una alternativa para los semáforos, estos son los semáforos binarios o **mutex** que veremos más adelante

Ambos tipos de semáforos contienen una cola para los procesos bloqueados y a su vez enfrentan la cuestión ¿Qué orden se debe tomar para su ejecución? Ambos utilizan la política *FIFO* (*First-In First-Out*) esta misma política convierte al semáforo en un **semáforo fuerte**. Por otra parte, un semáforo que no especifica el orden de la ejecución se le denomina **semáforo débil**.

Si bien no vemos a los semáforos de la misma manera que la vida real (3 estados para indicar avance), tenemos 3 posibles rangos de valores para indicar que acciones es capaz de tomar un proceso nuevo de entrada.

Los semáforos parecen una estructura muy sólida, pero presentan diferentes problemas, como la necesidad forzosa de implementar *semWait* y *semSignal* a nivel atómico. Pero su verdadera problemática radica en su misma fortaleza, la exclusión mutua incluso siendo funcional realiza mucha carga en el procesador pero que da pie a usar un esquema de hardware para la exclusión mutua.

- **Mutex:**

Tipo de semáforo que como su nombre lo indica, su valor oscila entre 1 y 0 para la ejecución de procesos. Así mismo cuenta con 3 operaciones propias.

- Inicialización: puede ser declarado en 0 o 1.
- Método *semWaiB:* comprueba el valor del semáforo. Si el valor esta asignado a 0 el proceso será bloqueado. Si el valor esta asignado a 1 permite al proceso continuar con su ejecución, pero cambiando el semáforo a 0.
- Método *semSignalB*: comprueba si existe un proceso bloqueado en el semáforo. En caso de existir, lo desbloquea. Si no hay procesos bloqueados, el semáforo será asignado a 1.

Los Mutex vienen como un intento por solucionar la exclusión mutua basado en la estrategia de semáforos, pero con una implementación más sencilla en comparación, la diferencia fundamental es el valor que el semáforo adopta y que al ser más limitado propone menos situaciones en la que el semáforo será sometido, además, en comparación representa un mejor rendimiento por tener que comparar un valor más simple.

- **Dekker:**

Es uno de los primero intentos de Dijkstra por resolver la exclusión mutua que contenía 2 procesos a la vez que intentarían acceder a 1 recurso protegido en su sección critica que inicialmente consiste en ser cortesivo entre sí, los procesos intentarán acceder a dicho recurso protegido por turnos que se alternarían según la utilización que el proceso contrario, es decir, se cederá el permiso a acceder a los recursos a otro proceso una vez que el proceso en turno finalice su utilización y turno. Esta es la primera propuesta del algoritmo que presenta problemas de rendimiento por qué la ejecución de los procesos se determina según la velocidad del proceso más lento mientras que el otro estará en espera activa para acceder.

La segunda alternativa busca generar una variable global que indica el estado del proceso que indicará cuando un proceso requiere entrar en sección critica y removiendo los turnos pero

representa un problema aun mayor, en esta solución no se garantiza la exclusión mutua, permitiendo que ambos procesos se encuentren en sección critica.

Durante su tercera revisión se hace un cambio de validaciones siendo que, si un proceso quiere acceder a su sección crítica y modifica su estado, ya no será posible que otro lo haga hasta que se liberé el recurso y en caso de hacer el cambio de estado a *verdadero* y ya estar ocupado el recurso, simplemente se boqueará el proceso que recién haya modificado su estado. Si bien se vuelve a garantizar exclusión mutua se genera un problema nuevo, es posible que ambos realicen un cambio de estado antes de la ejecución del ciclo interno del algoritmo y ambos permanezcan Inter bloqueados indefinidamente.

La cuarta alternativa propone hacer cortesivo entre los procesos que se involucran siendo que ahora los procesos conocen el estado del otro y permitiendo que el proceso alternativo acceda a su sección critica en su lugar, pero esto genera una situación que lo hace inviable, pues es posible que suceda un circulo vicioso que haga perder tiempo a procesador en el intento de acceso al recurso deseado. Y aunque este hecho no se de forma indefinida, su mera posibilidad lo hace descartable.

Finalmente, a solución explicada es una combinación del primer intento y el último intento haciendo que ambos procesos conozcan sus estados, pero solo alternando entre ellos cuando el turno se ha finalizado completamente.

Si lo ejemplificamos en la vida real tenemos la asignación de 1 juguete a 2 niños (**a** y **b** respectivamente. Cuando **a** quiera utilizar el juguete deberá primero saber si es su turno para utilizar el juguete, en caso de no serlo debe esperar hasta que sea su turno, pero si ya es su turno entonces puede utilizarlo (sección critica). Ahora sucede que **b** quiere usar ahora el juguete pero ya esta ocupado por **a** en ese caso igualmente debe esperar pero a la vez debe hacer saber que quiere utilizarlo, en este punto **a** reconoce que **b** quiere hacer uso del juguete pero el no ha acabado de hacerle uso, por ello, solamente cuando haya finalizado con su utilización, cederá el turno a **b**.

- **Peterson:**

En su solución se indica que teniendo un estado **P0** con estado *verdadero*, mientras otro proceso **P1** no podrá entrar a sección critica. También se evita el bloqueo mutuo de forma que, si **P0** se encuentra en su ciclo while dentro del algoritmo, pero bloqueado, esto representa que el estado de **P1** sea *verdadero* y el *turno* sea **1**, en este punto **P0** puede acceder cuando el estado de **P1** sea *falso* nuevamente o cuando el *turno* sea 0.

Esto permite lo siguiente:

- Los estados no pueden generar desinterés en su sección critica debidamente por su estado.
- Los estado no permanecen esperando su sección critica por que el *turno* los hace retomar su acceso.
- Los procesos tampoco pueden monopolizar el acceso por que están obligados a permitir que el siguiente proceso sea capaz de acceder a su sección critica.

- **Paso de mensajes:**

El paso de mensajes viene a solucionar la exclusión mutua a través de un proceso sincronizado y colaborativo, durante el paso de mensajes, los procesos interaccionan entre sí intercambiando datos entre sí.

Para su funcionamiento optimo y efectivo es necesaria la implementación primitiva de los métodos **send(destino, mensaje)** y **receive(origen, mensaje)**. Cada uno de estos métodos realiza la acción en su nombre y reciben como parámetros una proceso y el mensaje que se enviará o se recibirá.

Sincronización

Enfocándonos en **sincronización** sobre el **paso de mensajes** sobre dos mensajes destacamos que un proceso receptor solamente recibirá un mensaje hasta que otro proceso haya enviado alguno, por ello veremos primero el envío de mensajes.

El *envío* (send) genera dos posibilidades al hacer su ejecución, lo primero es que el proceso emisor se bloquea hasta que el mensaje enviado logra ser recibido, la segunda posibilidad no bloquea ni detiene al proceso.

De la misma manera, la recepción genera dos alternativas. La primera indica que si el mensaje se envío antes, se recibirá y se continuará con la ejecución, o su alternativa por otra parte determina que si no hay mensajes en espera el proceso puede bloquearse hasta que llegue un mensaje o continuarse ejecutando dejando de lado la recepción.

Con esta información sobre la mesa sabemos que tanto receptor y emisor pueden ser bloqueantes considerando 3 combinaciones típicas aunque en realidad solo se vean comúnmente en implementación 2 de ellas, siendo más naturales las que involucran un envío no bloqueante. Las 3 alternativas relevantes son:

- **Envio bloqueante, recepción bloqueante:** El receptor y el emisor se bloquean hasta confirmar la recepción del envío.
- **Envio no bloqueante, recepción bloqueante:** El proceso emisor puede continuar su ejecución, pero el receptor debe bloquearse hasta recibir el mensaje.
- **Envío no bloqueante, recepción no bloqueante:** Ni emisor ni receptor se bloquean durante el intercambio de mensajes.

Direccionamiento

Durante el envío y la recepción de mensajes ocurre un evento responsable de hacer llegar los menajes que consta de diferentes estrategias según las necesidades, pues hasta ahora hemos visto el **direccionamiento directo** que especifica los procesos destinatarios. En el **direccionamiento directo** existen 2 alternativas para la recepción de mensajes, una de ellas indica que se debe especificar al proceso emisor/origen, por tanto se debe conocer anticipar el proceso que espera al mensaje, es una buena alternativa para procesos cooperativos, pero no es la solución definitiva, existe la segunda solución que no especifica directamente el origen, que entre otros, es útil para procesos que requieren la espera de confirmación

Además del **direccionamiento directo** se encuentra el **direccionamiento indirecto** como se indica, no se especifica directamente el emisor ni el receptor, pues no hay interacción directa entre ellos, en su lugar toman un medio intermediario para completar su comunicación, este medio se le puede conocer como *buzón*, sitio en el que se depositan y recolectan respectivamente los mensajes dependiendo del rol que cumpla el proceso en el momento. Esta estrategia de intermediario da a

pie a muchas posibilidades de implementación y su principal ventaja es que permite flexibilidad en el envío de mensajes. Lo más destacable de sus ramificaciones son las relaciones que se generan siendo**:**

- **Uno a uno:** Para relaciones privadas entre procesos que evita la interferencia entre otros procesos.
- **Muchos a uno:** Permite relaciones cliente servidor donde el servidor proporciona información a otros procesos cliente, en este caso el buzón es conocido como *puerto*.
- **Uno a muchos:** Un solo emisor puede tener diferentes receptores para difundir el mensaje.
- **Muchos a muchos:** Contiene varios procesos como servidores concurrentes a la par de diferentes clientes.

Dependiendo de la elección se asignan buzones de manera dinámica o estática, pues los buzones son generados por los receptores que además son los dueños de dicho elemento. Cuando se elimina el proceso propietario, el *buzon* también se eliminará con el, aunque existe la posibilidad de que el buzón pertenezca al sistema operativo.

Formato de mensajes

Es necesario que los mensajes cumplan con cierta estructura para garantizar su funcionamiento, una estructura de mensaje general es la de la siguiente ilustración, si bien es un formato general, es posible generar variaciones a conveniencia.

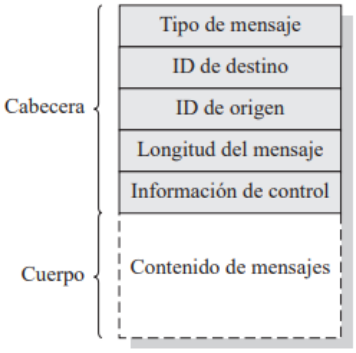


Figura 5.19. Formato general de mensaje.

Si bien el mensaje puede contener todo tipo de información, es importante cuidar el rendimiento, por ello algunos diseñadores prefieren optar por mensajes cortos con longitud fija para entre otras cosas, reducir la sobre carga del procesamiento y el almacenamiento.

Sobre todo esto no debemos olvidar el orden de solución, siendo *FIFO* la más sencilla de manejar pero por la naturaleza de prioridades, es difícilmente la mejor elección para esta estrategia, pues durante la ejecución del sistema, se generarán mensajes que son más prioritarios que otros, para esto se puede permitir al receptor un analisis de la cola de mensajes para determinar con cual seguir.

Finalmente ejemplificaremos el problema del productor y del consumidor, aprovechara muy bien los buzones para sustituir el uso de *buffers* indirectamente. Los buzones utilizarán la cantidad de mensajes contenidos para dar luz verde al cada actor a realizar sus debidas acciones, En el caso del productor genera entradas al buzón *puedeConsumir*. Si existe mensaje en este buzón, el consumidor entrará en acción. Por otro lado, el buzón *puedeProducir* disminuye su tamaño y crece con cada consumo.

Esta solución permite contener diferentes productores y consumidores siempre y cuando tengan acceso a los buzones necesarios

```
/* programa productor consumidor */
const int
     capacidad = /* capacidad de almacenamiento */;
     null = /* mensaje vacío */;
int i;
void productor()
{    message pmsg;
     while (true)
     {
          receive (puedeproducir, pmsg);
          producir();
          receive (puedeconsumir, pmsg);
     }
}
void consumidor()
{    message cmsg;
     while (true)
     {
          receive (puedeconsumir, cmsg);
          consumir();
          receive (puedeproducir, cmsg);
     }
}
void main()
{
     create_mailbox(puedeproducir);
     create_mailbox(puedeconsumir);
     for (int i = 1; i <= capacidad; i++)
          send(puedeproducir, null);
     paralelos (productor, consumidor);
}
```

Figura 5.21. Una solución al problema productor/consumidor con *buffer* acotado usando mensajes.

Ilustración 8 Implementación del productor consumidor con paso de mensajes

Conclusión

La exclusión mutua es un estado en la concurrencia de la programación muy útil para preservar la integridad de los datos compartidos en un sistema, gracias a ello podemos utilizar un sistema que se hace valía de valores compartidos sin problemas ni problemas de integridad, a lo largo del tiempo surgieron diferentes estrategias que lo que más resaltamos es la similitud que algunas soluciones tienen con la realidad y como es que la misma se puede aplicar a procesos en un contexto muy especifico, el ejemplo a tomar es el paso de mensajes, que en la vida real se asemeja a cualquier equipo de trabajo que realiza sus procesos con la comunicación como medio de trabajo, en el equipo se debe lograr la integridad del proceso a través de la solución para evitar malos entendidos y modificaciones no deseadas o planeadas, que se traduce en el acceso de lectura y/ escritura del recurso compartido en la exclusión mutua.

Bibliografía

(s.f.). Obtenido de open4tech: https://open4tech.com/rtos-mutex-and-semaphore-basics/

Stallings, W. (2004). *Sistemas operativos- Aspectos internos y principios de diseño.* Madrid: Pearson educación.